[1]Similarity in Computer Programs
(draft version to be presented to 107-441B (Philosophy of Science II), McGill University course Winter 1999 semester)

We have all used computer programs that behave, look, act and so on alike. Yet despite this intuitive recognition and legal cases precisely concerning this issue, there is no generally accepted standard for comparing computer programs for similarity[1]. In this presentation this issue will be explored and some hints in this direction that may prove useful shall be suggested.

This presentation will consist of 4 parts. In the first part, we shall find a brief discussion of what a philosopher of technology is, and further why similarity is in part a philosophical concept, and finally an explanation why computer professionals should actually care about such issues. In the second part proposals for working definitions of functional similarity, algorithmic similarity, look-and-feel similarity and feature similarity will be made and explored with use of examples. For those of you who do not have familiarity with the programming languages and techniques being looking at, indulgence is begged and it is hoped that the examples are sufficiently clear. Section three of the presentation will show how many of the "barriers" erected around the different kinds of similarity are in fact artificial, and hence the different kinds of similarity can often collapse into one another. Finally, a brief look at the ethical and legal aspects raised by the previous considerations, followed by a short remark concerning recommended policy (policies) based on these.

Part 1

On to the first part, then. Let us begin by briefly exploring exactly what a philosopher of technology is by breaking this strange phrase down into its components. First: "philosopher" means anyone who works at general problems of knowledge, being, the good, the right, and so on. Secondly, technology consists of artifacts designed through the aid of science. (This distinguishes it from "craft", as well. See Bunge 1998 for more details.) Since computer science and technology centers around such artifacts (despite the fact that many times it seems computers were designed with voodoo!) a philosopher of technology may study the general features of reality, knowledge as they pertain to questions raised by computing. A philosopher is concerned with similarity because it is a very general feature of things. In fact, exploration of things spoken of generally is in the domain of philosophy known as ontology, or metaphysics (Bunge 1977). These two words are to be regarded as synonyms, and please note that they are being used in their philosophical sense in the present work[2]. This means that the issue of similarity in computer programs is in part a philosophical question. However, it is not wholly a philosophical issue for several reasons. One, the most important reason, is that the computers we use are artifacts, as has been mentioned, and hence are designed and programmed by human beings. This means philosophers must keep in mind the foibles of humanity when discussing this issue of similarity and particular the foibles of the computer program's designers. We shall see more on this anon. A secondary reason that the question isn't only philosophical is because deciding on criteria for similarity in computer programs can lead to value judgements, which in

1 Or, slightly more correctly, the present author has not encountered any such thing. A few searches on the ACM website, as well as a literature perusal in the McGill University Physical Science and Engineering library turns up nothing.

2 "Metaphysics" does have a religious meaning and a everyday meaning, which are not being used here.

turn can lead to policy making, which is another technology. Computer scientists should care for these issues, because it seems plausible that they very much want to get appropriate credit for their hard labour, and don't want to be ripped off by a "similar program from a competitor" or the like. Even if getting ripped off or credit isn't the issue, perhaps a desire to create something unique is, or something similar but better. Or to replace some software piece with something "that does the same job". Without the concept of similarity rendered precise, when do we want to say these goals have or haven't been accomplished? or that desires have or have not been frustrated or compromised?

Before we can begin the exploration of what similarity in computer programs amounts to, I will first provide a definition of computer program and a few related issues, following the 3rd edition of Knuth's celebrated *The Art of Computer Programming*. He suggests (pg. 5):

> "An expression of a computational method in a computer language is called a program."

This definition is of critical importance, because it masks a dangerous equivocation that some computer scientists (see, for example, Cormen, Leiserson & Rivest 1996) have fallen prey to. This equivocation centers around the definition of the term algorithm, which to some will be conspicuously absent from Knuth's definition. Some computer scientists use algorithm to mean what Knuth means by computational method, so let us examine the distinction here. Knuth's meaning of algorithm, which is closest to the way it is used in mathematics (see Machover 1997 or Bunge 1999), is as follows.

A finite sequence of operations for solving a specific type of problem with the following features:

- 1) it always terminates after a fintie number of steps (*finiteness*)

- 2) each step must be precisely defined and the actions to be carried out must be rigorously and unambiguously specified for each case (*definiteness*)
- 3) it has zero or more quantities that are either given to intially before it begins or dynamically as it runs (*input*)
- 4) it has one or more quantities that have a specified relation to the inputs (*output*)

- 5) it is desirable that the operations in the algorithm must be sufficently basic that they can be done exactly and in a finite time (*effectiveness*)

Note that a computational procedure does not have the finiteness property, whereas an algorithm in Knuth's sense does. Euclid's "greatest common measure" procedure is one such computational procedure. This procedure does not terminate if the lengths being compared are incommensurable. A more common example from computer science proper is any example of a reactive process. That is, any computational procedure that continually interacts with its environment, such as an operating system. Criterion five has been separated from the rest for another reason, namely because it is a little vague. Intuitively, it is desirable that each step be performable by a human exactly and in a finite time. This is a little problematic, as it is unclear whether one wants to say "in principle" or "in practice". Each choice creates problems that are tangental to our present purpose, so they will be ignored.

But note very well that the domain of values for the inputs and outputs is not specified. In principle they can be anything. Also note that the effectiveness property means that there can be other computational processes within a given process, including, of course, algorithms which can be viewed as a special case. Hence a computer program is a hierarchy of computational processes specified in a precise, formalized language. Languages in this context do include a syntax and a semantics. In our discussion of similarity to follow, semantics will be of greater importance. It will be important in order to make accurate comparisons, so be warned. For those of you who don't know, exact, rigorous computer programming language semantics is a very technical field; the presentation will not rely on any knowledge of this field.

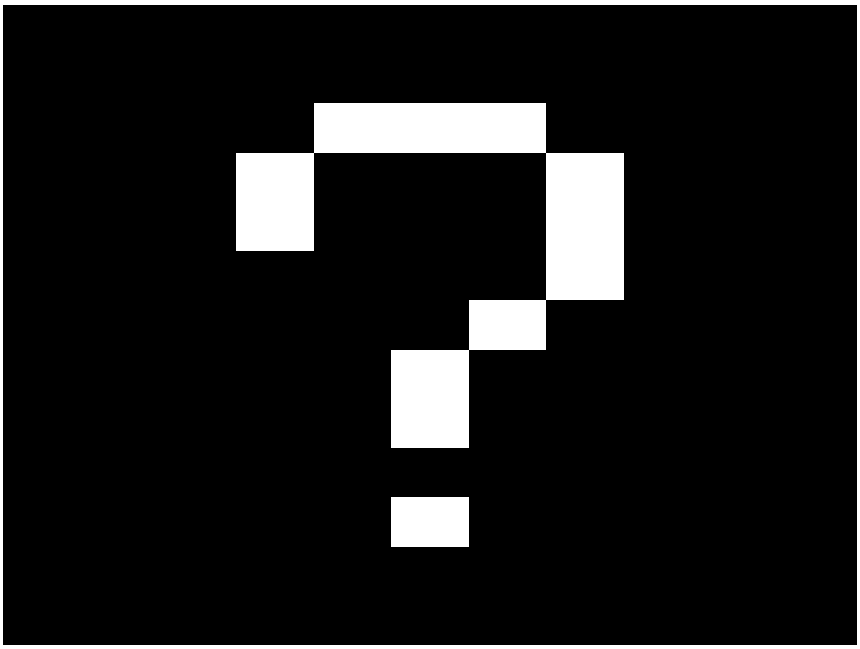
Part 2

Now that we have seen the sort of "background" to this enterprise, let us now turn to our first look at the kinds of similarity. Before continuing, a note concerning a simplification being made. It is assumed (by adopting a modification of Knuth's definition) for the moment that a computer program consists the collection of its source code, including that of any internal or external libraries, and its resources or equivalent, in some suitable form, together with its "application" or binary as the case may be. We shall see why in due course why simply either source code/resource code or the application alone is insufficient. This pseudodefinition will create a few problems later, but for now it will suffice. (Introducing a more rigorously correct definition at this stage could poison the well in favour for or against several positions and issues we shall encounter later.) One may note that this definition deliberately leaves vague the notion of source code.

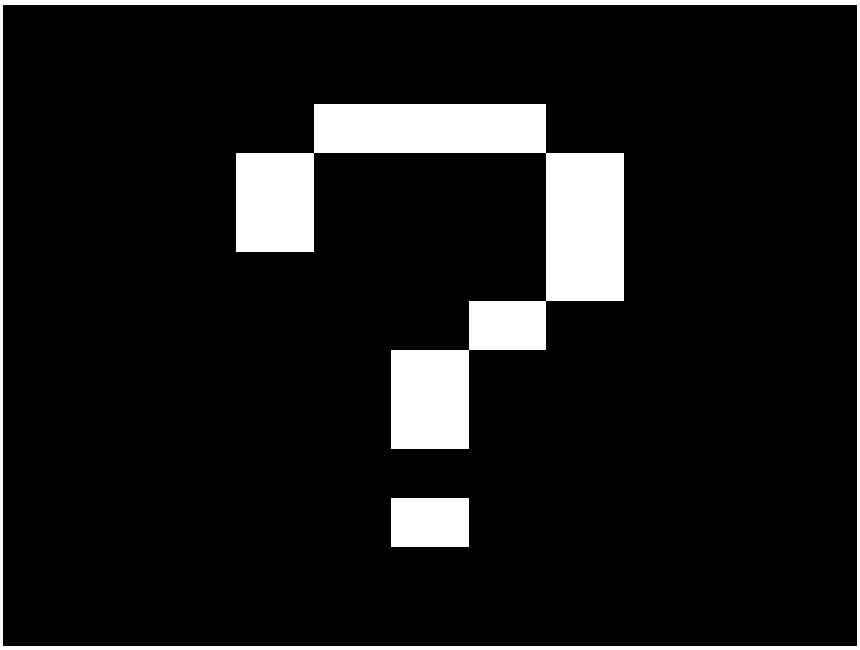
So, then, in this section, I will propose 4 kinds of similarity amongst computer programs and illustrate them with examples. The 1st kind of similarity to explore is functional similarity. The intuition here is that the programs in some sense "do the same thing." This notion is a bit blurred and leads to all kinds of sort of "grey areas", but let us look at some examples of what this "first intuition" tells us. Our first example is that of two word processing programs. Here's a screenshot of a very old word processor from the Apple II (Apple Writer).

<picture>

Here's another, more modern one:



This is Nisus writer, which was used to write this presentation. Both of these programs allow the user to enter text, modify its layout on the page, underline, and so on and so forth. However, Nisus Writer has all the “modern” features, such as button bar, pull down menus, etc. Apple Writer doesn’t even support mice! But how similar are either of these two with BBEdit Lite, here?



BBEdit Lite is not normally classed as a word processor because it doesn’t allow changes of font, spelling check, and so forth - instead we know it as a text editor. But these are features that Apple Writer didn’t have 15 years ago either. Even today, there are borderline cases, for

instance, x-emacs. Is x-emacs a word processor or a text editor? Can it be both? These raise questions of functional similarity.

One possibility would be to simply catalog a set of functions of each program and compare with a set-theoretic measure. This suggests Formula 1:

$$S_f(P_a, P_b) = \frac{\overline{F_a(P_a) \cap F_b(P_b)}}{\overline{F_a(P_a) \cap F_b(P_b)}}$$

where S_f is the functional similarity, F_a is the set of features of program P_a and F_b is the set of features of program P_b. It ranges from zero, when the programs have no features in common to unity when the programs are feature-identical. Note that this quantity is dimensionless. (Which is just as well, because inventing an appropriate scale and units is beyond the present author at the current time.)

But this can't quite be right. For one thing, the idea of a set of functions for a program is very ill-defined. Does one count "ability to make styled text" count as a function? or "ability to use the 'standard' bold/italic/underline"? Also note that you cannot easily restrict comparisons of two very unalike programs, so even very different programs will have non-zero similarity. For instance, both Photoshop and Nisus Writer allow for creation of text, but intuitively these programs are very different functionally in virtually every respect. This exactification is promising, as it does suggest that all one needs to do is to move from similarity to a notion of a feature. Hence a working definition is found here.

A feature of a computer program =df a single atomic action that may be undertaken by a user (human or another program or piece of hardware) to provide or transform the input to the program, to change the functionality of the program itself, or to display documentation.

The latter clauses allow for preference changing features as well as help features. The "provide input to the program" clause is built in as well, as it is felt it is appropriate to consider various methods of getting input in to be separate. For instance, drag and drop, command line argument, and so on are all different ways. With this definition of feature, it is hence possible to use formula 1, proposed above. (Note that this definition still has problems particularly with programs such as games and the like, as well as with easter eggs.)

The next kind of similarity that should be explored is that of algorithmic similarity. This one is fairly easy to understand, but requires looking at the program's source code. After all, as we all know, many algorithms can produce "the same output". (Of course, what exactly we should take to be a modern computer program's output is a very thorny issue that we shall return to a bit later in the look and feel section.) Consider the following source, in the languages Scheme and C. All the first does is define a function to compute factorials recursively. Mathematically, the C function that follows is very similar. It is not completely identical, however. Exercise to complete before seeing the next section of the paper: Why aren't the two identical?

```
(define factorial
   (lambda (n)
     (if (= 0 n)
       1
```

(* n (factorial (- n 1))))))

```
int factorial (int n)
{
      if (0 == n)
            return (1);
      else
            return (n * (factorial (n-1)));
}
```

Both factorial functions are recursively defined. Both "bottom out" (have base case) of zero, as they should. There are in fact several distinct differences despite the very strong algorithmic similarities here. First and foremost concerns maximum sizes of data types. As some of you may know, Scheme has arbitrarily sized integers. One can use the first function and calculate factorial 100, or even 1000. The C function is of course going to actually "misbehave" beyond a certain point due to integer rollover. In fact, our putative similarity is worse than that, because the sizes of ints is implementation defined, so we have to know something about the host compiler and its settings to know how similar two programs are algorithmically. Secondly, error checking is completely different and, again, external to the function. In the case of finite mathematical functions - i.e. ones that compute and return specific numerical values[3], a very simple metric of similarity can be used. In this case, the following similarity function, formula 2, is proposed:

$$AS_{ab} = \frac{\overline{IO_a \cap IO_b}}{\overline{IO_a \cup IO_b}}$$

where AS_{ab} is the similarity between a and b, IO_a is the set of ordered pairs <input, output> for function a, IO_b is the set of ordered pairs <input, output for function b> and the bar represents cardinality of a set. Thus the function ranges between 0 for no similarity to 1 for complete similarity. But this won't do for all cases of algorithmic similarity even in the restricted case of algorithms that compute specific numerical values. It is probably best to consider the above formula a first order approximation. To see this, consider the following two C function collections (purists should assume that appropriate typedef struct / typedef enum declarations and the like are elsewhere):

collection 1:

```
void QuickSort (Array A, int m, int n)
{
      int i,j;

      if (m < n)
      {
```

[3] It may be argued that all the data computers (or Turing machines) deal with are really integers and hence all functions are of a numeric sort. Two objections can be made to this account. One is simply that it is false that computers deal with integers only. See Clements 1994. Even if this is not granted, it should also be noted that if one represents, say, an anagram solver, as a numerical calculation (given some translation rule ['A' = 65, and so on], this will mask the similarity that one would expect. As after all, this would appear to collapse all uses of computers into one, which seems unsatisfactory.

```
            i = m;
            j = n;
            Partition (A, &i, &j);
            QuickSort (A, m, j);
            QuickSort (A, i, n);
      }
}

void Partition (Array A, int *i, int *j)
{
      Type Pivot, Temp;

      Pivot = A[(*i + *j) / 2]; /* This QuickSort uses middle item as pivot */
      do
      {
            while (A[*i] < Pivot) (*i)++;
                  /* find leftmost i such that A[i] ≥ Pivot */
            while (A[*j] > Pivot) (*j)++;
                  /* find rightmost j such that A[i] ≤ Pivot */
            if (*i <= *j) /* if didn't cross, swap */
            {
                  Temp = A[*i]; A[*i] = A[*j]; A[*j] = Temp;
                  (*i) ++;
                  (*j) --;
            }
      } while (*i <= *j); /* while they haven't crossed each other */
}
```

collection2:

```
void BubbleSort (Array A)
{
      int i;
      Type Temp;
      Boolean NotDone;

      do
      {
            NotDone = false;
            for (i = 0; i < n-1; i++)
            {
                  if (A[i] > A[i+1])
                  {
                        /* exchange A[i], A[i+1] to put them in sorted order*/
                        Temp = A[i]; A[i] = A[i+1]; A[i+1] = Temp;
                        /* since we swapped, need another pass */
                        NotDone = true;
                  }
            }
      } while (NotDone);
}
```

This example should drive home how much of a rough estimate the first formula for

calculating algorithmic similarity was. After all, the two sorting methods written out above are almost diametrically opposed in many ways, yet produce the same output. Output alone cannot therefore be a measure of algorithmic similarity.

Analysis of the BubbleSort algorithm in terms of n, the size of the array to be sorted, yields a best case running time of $O(n^2)$, whereas the QuickSort algorithm yields O(n log n). Both, however, are $O(n^2)$ in the worst case. Further, in fact, there are causes of these running time differences that are apparent only to one with the source code, in this case. What is suggested in this case is a correction factor to the initial expression for both best case running time and worse case running time. Thus to analyze algorithmic similarity the programs' source code must be available. But comparing running times is difficult; there are obviously at least a countable infinity of possible running time classes. Books on algorithms often list "typical" ones, such as O(1), O(log n), O(n), O(n log n), $O(n^2)$, $O(n^3)$, $O(2^n)$ and so on. But these are only a finite subset of values. Algorithms with running times other than these typical cases are well known. For instance, the running time of Strassen's (1969) algorithm for the multiplication of n by n matrices is $O(n^{\log 7})$. A suggestion at this stage would be to calculate the specific running time of a program by using a specific n, called n*, and then comparing. This has two interrelated flaws, both centering around the choice of the n*. Firstly, due to the constant factors which are dropped in O notation, the n between certain values is "not typical" For instance, if we have two running time functions $r_1(n) = 50 \text{ n log n}$ and $r_2(n) = 2n^2$, for values of n less than 41, the program with running time function r_1 will perform worse, despite how r_2 will eventually do so. This is masked by the "O notation hierarchy conventions". Also note that this relates to the second issue - how could one make a principled decision on the value of n that would be appropriate for all cases. A suggestion, which shall be adopted here, will be to let n be arbitrarily large. A specific value will be suggested below. This stipulation means that programs are compared in algorithmic similarity with arbitrarily "large input", despite the fact that it may prove useful to compare at some other point. This is the usual convention with running times anyway, but it will be important to keep this in mind when shortcomings and possible refinements are discussed below.

Thus all that has to be done is to find some way of producing an cardinal ordering of all possible running time classes. This is easier said than done, as we have seen previously. Standardizing the RUNNING time would help here. Then one can compare "ns" from the O(n) notation, assuming that given this running time the n is sufficiently large that the constant factors which the O(n) removes are insignificant. If we look at a typical table of running times, such as the following, which is adapted from Standish 1995:

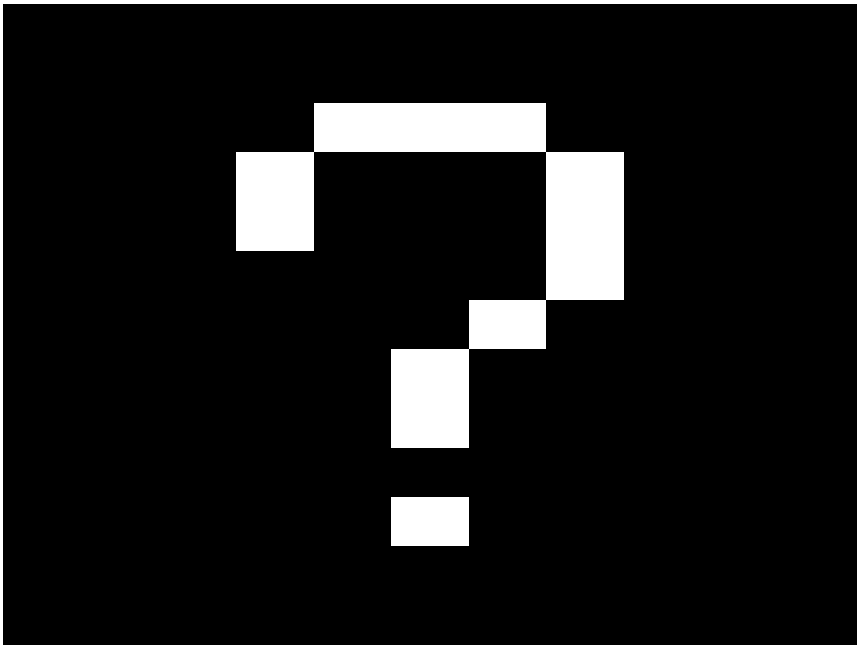
Algorithm A stops in f(n) microseconds					
f(n)	n = 2	n = 16	n = 256	n = 1024	n = 1048576
1	1	1	1	1	1
$\log_2 n$	1	4	8	1	2
n	2	16	256	1024	1048576
$n \log_2 n$	2	64	2.05×10^3	1.02×10^4	2.10×10^7
n^2	4	256	6.55×10^4	1.05×10^6	1.10×10^{12}
n^3	8	4096	1.68×10^7	1.07×10^9	1.15×10^{18}
2^n	4	65536	1.16×10^{77}	1.80×10^{308}	6.74×10^{315652}

Clearly it will be difficult to find a standardized amount running time as what is a large or small running time varies tremendously. It is suggested the algorithms have their running time complexity class determined in big-O notation, and then found an approximate line on this table. (Exponential running time algorithms are usually avoided like the plague, so it is fair to class them together.) Simply count the difference in rows and divide by the total number of rows in the table. Since best case/worst case/average case is often a concern, it is further suggested that this be done for each of the worst case and average case. (We do not usually care for best cases, after all.) For instance, quick sort and bubble sort are 1 different in the best case (O(n log n) vs. $O(n^2)$) and no different (on this account) as to worst case. So the similarity here is 6/7 x 7/7 x 1.000 (assuming that both are implemented on the same compiler or interpreter and run on the same host, etc.) for a total similarity of ~0.857. On the other hand, if we wanted to know the similarity between a sorting method that is able to run in linear time, for instance, radix-sort, to bubble sort, proceed as follows. First note that they run in 2 classes different in each case, so similarity is 5/7 x 5/7 x 1.000 = 25/49 =~ 0.510 similarity.

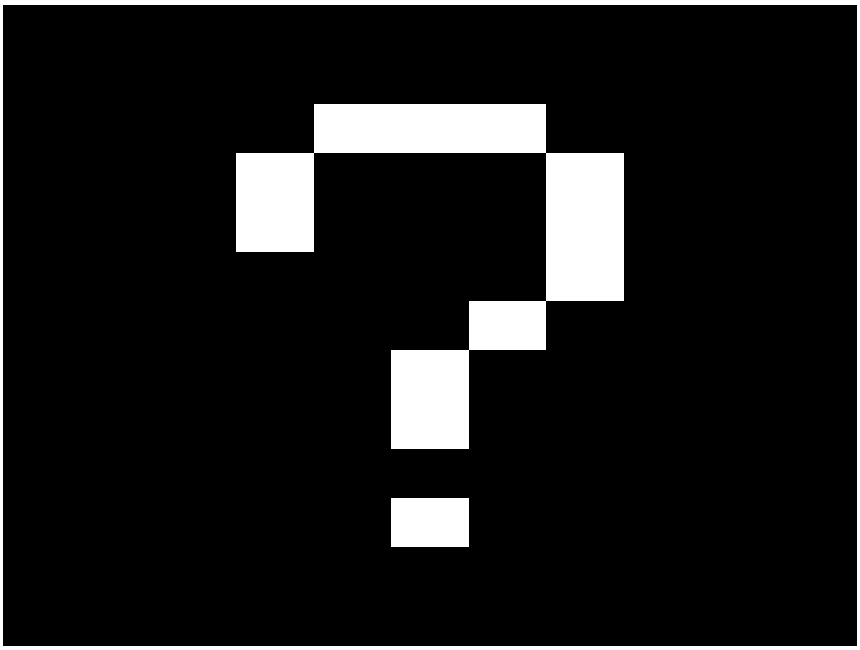
This approach allows for refinements in two directions. One is that we can get greater precision in our measure of similarity if we use a bigger table of standard running time classes. The second refinement is that if we temporarily relax the conventions of the big-O notation, we can see if a given value of n “really” falls into the appropriate class the notation suggests, and hence adjust one’s similarity measure by an appropriate amount for a specific case. (In this case, the similarity of a program to another depends somewhat on its input. We have seen hints at this previously when the C function to compute factorials was compared to the Scheme one previously.)

Now we can move on to look and feel similarity. Here’s where things get really dicey.

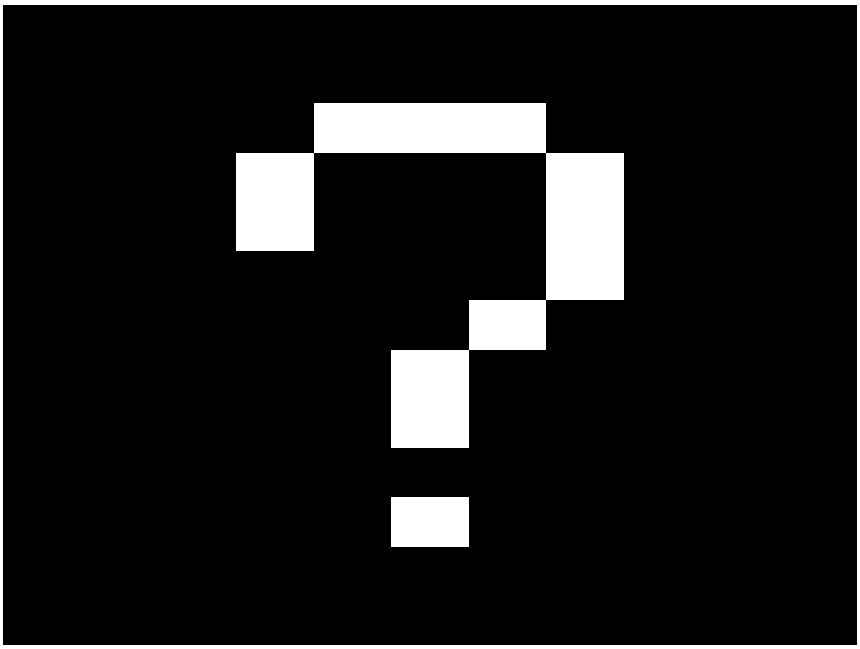
Here are several screenshots from various graphical interfaces.



Finder



Explorer

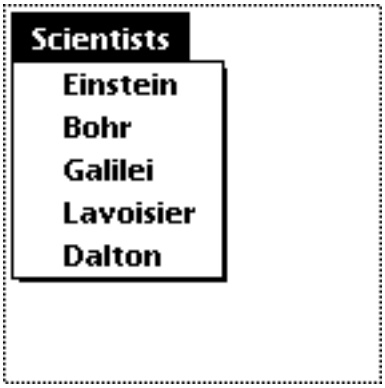


X11 with the twm window manager, slightly truncated.

Since they are all GUIs, they all share certain “look and feel” similarities, or so the intuition goes. For instance, the scroll bars are said to be similar this way. Of course, scroll bars behave somewhat differently in each case - whether they are accelerated or not, whether they are “proportional” or not, whether they perform “live” scrolling or not, and so on. These all relate to the “feel aspect”. One can also see that they look somewhat different. We also know that the user of the system has some control over such appearances, and in some cases feel, too. (Some systems allow the user to decide between proportional and non proportional scrollbars, for instance.) Note also in this case that look can serve to help indicate feel. A simple way to determine look and feel similarity would simply be a subjective one - ask users if they feel they are doing the same sorts of things and seeing the same sorts of items. This is of course problematic for all the usual reasons. If there is any objective look and feel, this can’t be the way to find it out generally, though it may give hints in the right direction. One possible solution, at least under MacOS alone, would be to dump the interface elements using DeRez and match up that way. The output of this procedure with two different menus where they differ simply by a separator line follows. (See the pictures of the menus.)

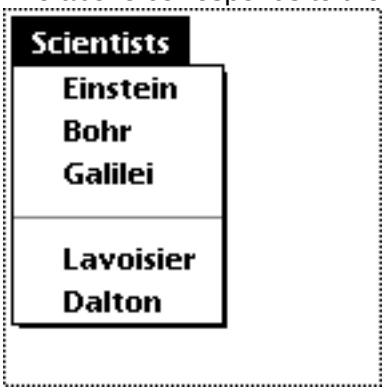
```
data 'MENU' (128) {
       $"0080 0000 0000 0000 0000 FFFF FFFF 0A53"            /* .Ä.............S */
       $"6369 656E 7469 7374 7308 4569 6E73 7465"            /* cientists.Einste */
       $"696E 0000 0000 0442 6F68 7200 0000 0007"            /* in.....Bohr..... */
       $"4761 6C69 6C65 6900 0000 0009 4C61 766F"            /* Galilei....ΔLavo */
       $"6973 6965 7200 0000 0006 4461 6C74 6F6E"            /* isier.....Dalton */
       $"0000 0000 00"                                       /* ..... */
};
```

The above corresponds to the following menu:




```
data 'MENU' (129) {
       $"0080 0000 0000 0000 0000 FFFF FFEF 0A53"            /* .Ä.............S */
       $"6369 656E 7469 7374 7308 4569 6E73 7465"            /* cientists.Einste */
       $"696E 0000 0000 0442 6F68 7200 0000 0007"            /* in.....Bohr..... */
       $"4761 6C69 6C65 6900 0000 0001 2D00 0000"            /* Galilei.....-... */
       $"0009 4C61 766F 6973 6965 7200 0000 0006"            /* .ΔLavoisier..... */
       $"4461 6C74 6F6E 0000 0000 00"                        /* Dalton..... */
};
```

The above corresponds to the following menu:



This suggests a first approximation to the “look” part of look and feel similarity. It is of course very crude because it doesn’t deal with cross platform issues at all. Nevertheless, I shall mention this approximation here. First, the “look” relation being defined is only pairwise. A menu looks like another menu, as in the example above. (We shall see the trouble it gets one into if one compares a menu to say, a window) Second, one must pad the smaller resource with extra bytes of a “undefined” sort to make it equal in length to the other. Then, form the list of all bytes in each resource. Call these R1 and R2, then using the expression below one can work out a crude “look ratio”. The “undefined” bytes are different from any other bytes. “List intersect” is like set intersection only that it is order dependant. (0 1) is maximally different from (1 0). Hence we have formula 3:

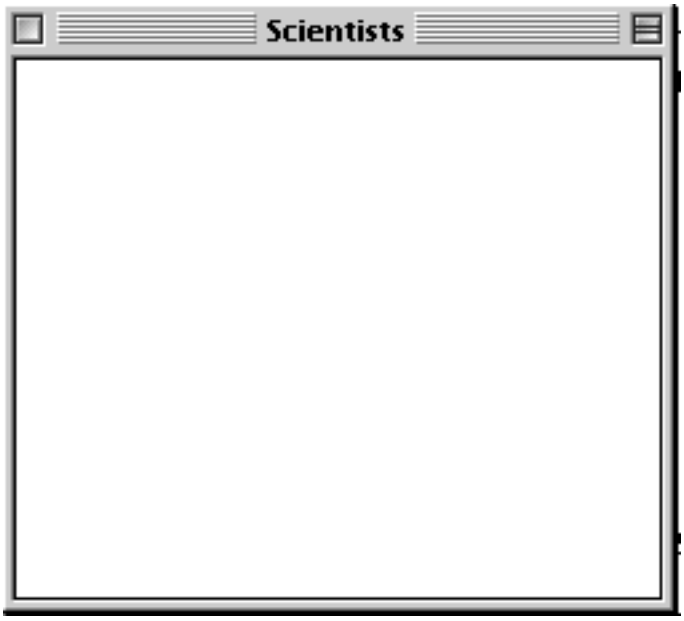
$$S = \overline{R1 \text{ (intersect) } R2} \,/\, \overline{R2}$$

Note that R1 could be used in the denominator as it always equals R2.

It is however EXTREMELY important to only compare resources of the same type. Let’s look at the comparison of the above two menu items, then compare one of them to a WIND

resource just to show how a “false similarity” can be found. First step is to pad MENU id 128 as it is 6 bytes shorter. Then count off similarities. 15/16th first line; 16/16 second; 16/16 third; 11/16 fourth; 0/16 fifth; 0/11 sixth. Hence the similarity is 15+16+16+11+0+0 / 16+16+16+16+16+11 = 59/91 = 0.648 similarity. If we compare MENU 128 to the following WIND resource (also ID 128) ... Note that I have deliberately titled the window “Scientists” for diadictic purposes.

```
data 'MENU' (128) {
       $"0080 0000 0000 0000 0000 FFFF FFFF 0A53"            /* .Ä.............S */
       $"6369 656E 7469 7374 7308 4569 6E73 7465"            /* cientists.Einste */
       $"696E 0000 0000 0442 6F68 7200 0000 0007"            /* in.....Bohr..... */
       $"4761 6C69 6C65 6900 0000 0009 4C61 766F"            /* Galilei....ΔLavo */
       $"6973 6965 7200 0000 0006 4461 6C74 6F6E"            /* isier.....Dalton */
       $"0000 0000 00"                                       /* ..... */
};
```




```
data 'WIND' (128) {
       $"0028 0028 00F0 0118 0000 0100 0100 0000"            /* .(.(............ */
       $"0000 0A53 6369 656E 7469 7374 73"                   /* ...Scientists */
};
```

The similarity here, naively, is 3/85 = .0349. Clearly, though, the menu and the window do not even have the same look, never mind feel. This just imposes a constraint on our first approximation. Can we quantify the feel, then? Intuitively, assuming that the items on each of the menus produced the same results in each case, we would say that the feel of the menus was very similar, as only the separator line distinguishes the two.

Comparing look similarity across platforms is more difficult. Superficially comparing bitmaps of screen dumps is bound to prove unhelpful for the reasons we saw in the discussion of numerical algorithms previously. For instance, consider three different interfaces. Two of these

are in one environment or program, but use radically different colour schemes. A third is in a different program of the same type (say Explorer and two colour-schemes in Finder.) but with a colour scheme like one of the first. Byte-by-byte comparison may very well lead to greater similarity between the two with similar colours despite the fact that the structure and look is very different. This should be avoided.

Further, comparing “feel” similarity is difficult. At the current moment, the present author does not have a good method for cross platform comparisons of look-similarity. However, there is a way to develop a measure of feel similarity. It is remarked that we appear to relate feel in some sense to functions and to features. Hence, it seems plausible to collapse the two. We shall see this in due course when we have looked at feature similarity in greater detail, which we shall turn to now.

Feature similarity is another area in which one intuitively has a feeling for similarity in computer programs. For instance, we say that two programs share features, e.g. spelling check. It is important, at first glance, anyhow, to distinguish this from functional similarity. Functional similarity is more of a similarity of results than similarity of process, which I regard feature similarity to refer to. What exactly counts as a feature is of course problematic, and further there are two ways in which programs can be feature similar. They can have similar features, for instance, Metrowerks C and MrC can both have compiler optimizations. They can also have features that are themselves similar, as if the two compilers both share loop-unrolling optimizations. (You may at this stage legitimately wonder if this is really a distinct category from functional similarity - but one can see the difference by recalling interface differences. Using a commandline or a GUI tool can be feature similar, yet not function similar. A filter could be invoked with a commandline or by using slider-bars in a GUI.

Features are different from functions for several reasons. For one thing, features do not have to affect the output or the running directly. It is proposed that they be regarded as “ways of doing things.” For instance, “spelling check” is a function, but invoking it by selecting it from a pull down menu is a feature. Thus one cannot have features without functions. Hence the following measure of feature similarity is proposed (formula 3):

$$FeS = \frac{\overline{I_a(f) \cap I_b(f)}}{\overline{I_a(f) \cup I_b(f)}}$$

To use this formula, pair all the functions of program a with all the ways in which it can be invoked. Call this $I_a(f)$ and call the similar collection $I_b(f)$ for the program to compare it to, then calculate as above, where FeS is the feature similarity. Note very well that this makes feature similarity parasitic on the notion of functional similarity, as if a program has all the same functions as another (say, in the case of two compilers, compile-mac68k, compile-macpowerpc, compile-windowsintel, compile-windowsalpha) and yet in program a these are invoked by pressing command-1, command-2, command-3 and command-4 whereas in another, they are invoked by passing -o1, -o2, -o3 and -o4 on the commandline, the programs have 1.000 functional similarity on these attributes, but 0.000 feature similarity. Note how fine grained we can make this, if wanted. We could compare, say, a Windows program, with its MacOS equivalent and note that the former uses control-P, control-S, control-O, and control-z for (say) print, save, open and undo, respectively. One measure of similarity feature-wise with the MacOS program would be 0.000, as it would use command-P, command-S, command-O

and command-Z for the same functions. However, if more precision was desired, one could compare each function component-wise and note a similarity of 0.500. On this measure, if the Windows program had, say, F3 for open and the rest as above, similarity would be 0.375. (Both of these comparisons assume (implausibly) that the program has no other method of doing those particular operations.)

This remark about fine-tuning the measure of similarity applies generally to all the kinds of similarity we have seen previously. One should always specify exactly what is to be counted prior to the calculation, a general lesson one should keep in mind in science, technology and philosophy.

This may begin to suggest a feel-similarity which has been promised. It shall be presented in the next section, which we shall turn to now.

Section 3

This section presents the feel-similarity, as it appears to further collapse the function-feature similarity connection we saw in the last section. It will also present several other connections between the sorts of similarity.

It was remarked earlier that feel-similarity appears similar to functional and by extension to feature similarity. Because the "feel" of something is generally regarded as something internal to the brain of a subject, and that "look" does affect feeling[4], it is moved that "look-and-feel" similarity is actually redundant. Hence look similarity together with feature similarity is what should be investigated in the cases where the traditional look and feel similarity is said to be at issue.

<More coming.>

Section 4

To conclude, several policy recommendations and related points. Firstly, similarity in computer programs is a very nebulous concept. Hence any policy or norm set up to regulate, guide, suggest or encourage certain kinds of activities must of necessity be somewhat open ended, at least at the time of writing. Secondly, it should be noted that there are many lacunae in the above formalization even taking into account the nebulosity. Thirdly, the nebulosity is hoped to be transitory. The present author is certainly of the opinion that these rough notions and intuitions can be honed further and perhaps a less opened policy can be recommended. He does not suggest at this stage that "similarity in all forms above x is too close and ought to be illegal or immoral." Finally, and related to all the previous general remarks, the calculation tools which have been sketched in the present paper are first attempts by the present author. He welcomes feedback and refinement by others, particularly by those whose skill and creativity in mathematics exceeds his.

---

[4] For instance, it is well known that various colours can produce varying kinds of emotional reaction, without any conditioning necessary. Chimpanzees, and to some extent, humans, are unconfortable in red light, to take one simple case.

References

Bunge, M. 1977. The Furniture of The World. Volume 3 of *Treatise on Basic Philosophy*. Dortrecht: Reidel.

Bunge, M. 1998. Philosophy of Science II: From Explanation to Justification. New Brunswick: Transaction Publishers.

Bunge, M. 1999. *Dictionary of Philosophy*. Amherst: Prometheus Books.

Cormen, T., Leiserson, C. & Rivest, R. 1996.*Introduction to Algorithms*. Cambridge: MIT Press

Clements, A. 1994.*68000 Family Assembly Languages* . Boston: PWS Publishing Co.

Knuth, D. 1997. Fundamental Algorithms. Volume 1 of *The Art of Computer Programming*, 3e. Reading: Addison-Wesley

Machover, M. 1996. *Set theory, logic, and their limitations*. Cambridge: Cambridge University Press.

Standish, T. 1995. *Data Structures, Algorithms and Software Principles in C* . Reading: Addison Wesley

Strassen, V. 1969. Gaussian elimination is not optimal. Numerische Mathematik, 14(3): 354-356